DBAF: Dynamic Binary Analysis Framework and Its Applications

Ting Chen(✉), Youzheng Feng, Xingwei Lin, Zihao Li, and Xiaosong Zhang

Research Center for Cybersecurity, University of Electronic Science and Technology of China, Chengdu 611731, China
{brokendragon,johnsonzxs}uestc.edu.cn,
fengyouzheng@gmail.com, xwlin.roy@gmail.com, gforiq@qq.com

Abstract. Dynamic binary analysis is difficult and burdensome. In practice, analysts always develop dynamic binary analyzers (DBAs) based on binary instrumentation tools (BITs), which are responsible for extracting information from a binary, monitoring or altering the execution of the binary. However, existing BITs either expose machine instructions to analysts or lack user-friendly APIs. Such problems result in a steep learning curve to grasp BITs and difficulties in eliminating bugs in DBAs. This work designs DBAF, a dynamic binary analysis framework that instruments binaries dynamically, conducts an online translation from machine code into an easy-to-handle intermediate representation (IR) and provides tens of APIs for IR processing. With DBAF, analysts can process binaries in the level of IR without the troubles to interpret machine instructions. Then, we develop five DBAs on top of DBAF, which are a division-by-zero protector, an IR counter, a memory tracer, a taint analyzer and a concolic executor. It demonstrates that DBAF can reduce the development effort for DBAs, especially the ones requiring semantic interpretation of instructions. Experiments show that DBAF brings about reasonable overhead in online translation.

1 Introduction

Binary analysis is a fundamental technique in many research fields, e.g., malware analysis, obfuscation/deobfuscation, software similarity analysis, and vulnerability discovery. It can be roughly classified into three categories, static analysis, dynamic analysis and hybrid analysis which combines static and dynamic analysis. This work focuses on dynamic binary analysis. Dynamic binary analysis is difficult and burdensome which needs rich experiences and considerable coding effort. In practice, analysts often develop dynamic binary analyzers (DBAs) based on existing binary instrumentation tools (BITs). With BITs, analysts can focus on the functionalities of the DBAs rather than the low-level details about how to load binaries into memory, parse binary files, extract information (e.g., control flow graph) from binaries, monitor or alter the execution of binaries.

However, exiting BITs have several shortcomings. First, some BITs (e.g., Pin [24], Dyninst [1], DynamoRIO [2]) expose machine instructions to analysts

M. H. Au et al. (Eds.): NSS 2018, LNCS 11058, pp. 361–375, 2018.
https://doi.org/10.1007/978-3-030-02744-5_27

directly, leaving analysts a complicated and error-prone process of semantic interpretation. Second, some BITs (e.g., Valgrind [28]) do not provide user-friendly APIs and documents, leading to a steep learning curve for analysts to grasp the BITs. Consequently, analysts have to search for the APIs of interest from less-clear documentations, example code developed by inexperienced programmers and even the huge source code of DITs.

This work designs `DBAF`, a dynamic binary analysis framework that instruments binaries dynamically, translates machine instructions into an easy-to-handle intermediate representation (IR). Moreover, `DBAF` provides tens of APIs, enabling analysts to process binaries in the level of IR. With `DBAF`, analysts do not need to interpret the semantics of machine instructions, and hence considerable development effort for DBAs can be saved. The implementation of `DBAF` is based on Pin and hence the invocation fashion of the provided APIs is similar with Pin's APIs. Besides, `DBAF` selects LLVM IR [20] as its IR format and reuse some code of Mcsema [10] for translation. Therefore, people who have experiences in Pin and LLVM can use `DBAF` without difficulties. We do not consider the requirement is an obstacle to use `DBAF` because Pin and LLVM are widely-accepted in both academia and industry.

To demonstrate the utility of `DBAF`, we implement five DBAs on it using the provided APIs. Three out of them are simple, which are a division-by-zero protector, an IR counter and a memory tracer. The code amount of them is comparable with those DBAs implemented on Pin directly. The taint analyzer and concolic executor are two complicated DBAs because they need to interpret the semantics of IR. The code amount of them is significantly lower than those directly implemented on Pin since the semantics of LLVM LR is much simpler than the semantics of machine instructions. Finally, experiments show that the translation process of `DBAF` incurs reasonable overhead.

In summary, the contribution of this work is threefold.

- This work designs `DBAF`, which instruments binaries dynamically and translates instructions into LLVM IR.
- `DBAF` provides tens of APIs, allowing analysts to handle binaries in the level of IR.
- We implement five DBAs on top of `DBAF`, using its APIs.

This paper is organized as follows. Section 2 focuses on the design of `DBAF`. Section 3 concerns the implementation of DBAs. Section 4 evaluates the translation overhead of `DBAF` and presents two practical cases about the taint analyzer. Section 5 reviews the related studies and Sect. 6 concludes.

2 DBAF

2.1 Overview

Figure 1 illustrates the high-level architecture of `DBAF` which takes in a binary, instruments it and then runs the instrumented binary. The workflow of `DBAF`

consists of six steps. Step one loads the binary into memory, parses the binary format and extracts relevant information. Then, DBAF fetches machine instructions from the binary, followed by the process of translation. The outcome of the translation step is LLVM IR. The instrumentor instruments the code of DBAs into IR and then IR is converted back to machine instruction in step five. Please note that step five is the reverse process of step three. Finally, step six runs the instrumented binary.

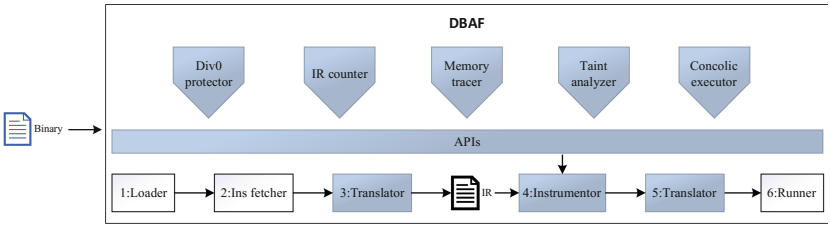

Fig. 1. High-level overview of DBAF

DBAF provides tens of APIs, allowing analysts to instrument binaries in the level of IR. Therefore, all the five DBAs invokes the proposed APIs without the troubles to interpret the semantics of machine instructions. The modules in Fig. 1 with the dark background are completely implemented by us (i.e., all DBAs and APIs) or adapted from existing BITs (i.e., translator and instrumentor), and the modules with light background directly leverage Pin. The code amount of DBAF has 6672 lines of C++, including 2258 lines for implementing the five DBAs.

2.2 Translation

The interpretation of machine instructions is burdensome and error-prone because an instruction set (e.g., x86) has hundreds of different instructions and many of them have complex semantics. Here taking a common x86 instruction cmp as an example, it has two operands which can be immediate numbers, registers and memory addresses. The bit-width of operands can be 8, 16 and 32. The execution of cmp does not change operands but affects flags. Which and how flags are affected are determined by the operands. For example, considering an instruction cmp eax, ebx where eax and ebx are two unsigned numbers, CF will be set to 1 if eax is smaller than ebx or 0 otherwise, and ZF will be set to 1 if the two operands are equal. Consequently, analysts have to spend significant effort to implement and debug complicated DBAs which requires semantic interpretation. For instance, Triton [34], a concolic execution framework that directly interpret x86/x64 instructions, has 35,120 lines of C++ code.

We propose to conduct an online translation from machine instructions into IR. An alternative is translating the binary into IR statically and then mapping instructions to IR dynamically. However, this method encounters similar

challenges that exist in static disassembly, such as data embedded in the code regions, variable instruction size, indirect branches [31]. Therefore, DBAF proposes online translation that fetches an instruction right before CPU executes the instruction, and hence DBAF overcomes the aforementioned challenges.

We select LLVM IR as the IR of DBAF due to its advantages. LLVM IR is a low-level RISC-like virtual instruction set, which supports linear sequences of simple operations like add, subtract, compare, and branch [19]. Therefore, the semantics of LLVM IR is much simpler than machine instructions like x86. Besides, LLVM IR is in three address form and strongly typed which facilitates program analysis and optimization [19]. In implementation, DBAF adapts Mcsema [10], which is a library lifting binaries into LLVM IR. As a static translation tool, Mcsema suffers from the similar drawbacks with static disassembly [31]. DBAF reuses the code from Mcsema [10] for lifting an instruction rather than a binary into LLVM IR, and hence it circumvents those drawbacks. Besides, we discover a bug in Mcsema resulting in an exception during translation due to a type mismatch. Mcsema accepted our suggestion and fixed the bug soon [13].

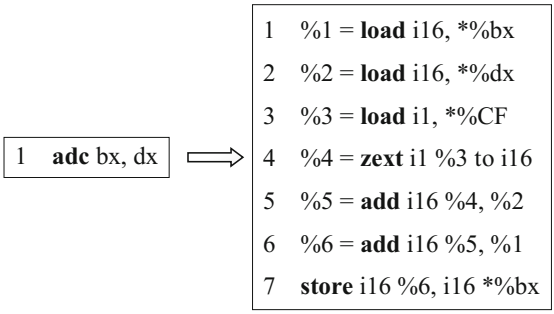

Fig. 2. Translate adc into LLVM IR

Figure 2 presents the translation of adc bx, dx into LLVM IR. The instruction adds dx to bx, and then add CF to bx. The LLVM IR after translation consists of seven statements. Statement 1 loads the value of bx into memory. %1 is actually a label representing load i16, *%bx. Therefore, one can simply think %1 is the value of bx. Similarly, statement 2 and 3 load the values of dx and CF, respectively. Statement 4 extends the 1 bit CF to 16 bits. The result of addition is represented by %6. Finally, statement seven stores the result into bx. The observation from this example is that the semantics of LLVM IR is much simpler than x86 instructions. Moreover, there are no implicit operands in LLVM IR; however, x86 instructions can have implicit operands (e.g., CF). Hence, the implementation of DBAs can be simplified after translation from machine instructions into LLVM IR.

2.3 API

Table 1 shows twenty representative APIs provided by DBAF. The APIs have similar invocation fashion with those APIs provided by Pin, so we explain some of them here. IR_AddInstrumentFunction() adds a function *func* to instrument in the level of IR, and hence *func* will be invoked after an instruction is translated into IR. Therefore, *func* can be considered as a callback function that processes each IR. IR_InsertCall() inserts a call to a function *func* before or after a specified IR, so that *func* will be called right before or after the execution of the IR. The two APIs IR_Ins() and IR_Address() map an IR to its corresponding instruction, so they bridge the gap between instruction instrumentation and IR instrumentation. IR_Opcode() and IR_Category() return the opcode and category of the IR, respectively. Please note that the opcode and category are defined in LLVM IR, rather than the instruction set.

Table 1. Twenty representative APIs provided by DBAF

API	Description
IR_AddInstrumentFunction	Add a function used to instrument at IR granularity
IR_InsertCall	Insert a call to a function relative to an IR
IR_Address	The instruction address of the IR
IR_ReadMemory	The address of the memory read by the IR
IR_WriteMemory	The address of the memory written by the IR
IR_Opcode	The opcode of IR
IR_OperandIsImmediate	Whether the specified operand of the IR is an immediate number
IR_OperandIsReg	Whether the specified operand of the IR is a register
IR_OperandIsTmp	Whether the specified operand of the IR is a temporary variable
IR_OperandTmp	Get the specific temporary variable of the IR
IR_OperandReg	Get the specific register of the IR
IR_OperandImmediate	Get the specific immediate number of the IR
IR_OperandWidth	The bit-width of the specified operand processed by the IR
IR_IsMemoryWrite	Whether the IR write memory
IR_IsMemoryRead	Whether the IR read memory
IR_Category	The category of the IR
IR_Ins	Get the instruction corresponding to the IR
syscall_entry	Call a function at the entry of a system call
syscall_exit	Call a function at the exit of a system call

LLVM IR can operate memory, registers, temporary variables and immediate numbers [23]. DBAF provides APIs to determine whether an IR read or write memory, whether an operand is an immediate number, register or a temporary variable, get the memory addresses, immediate numbers, registers or temporary variables operated by an IR. IR_OperandWidth() is responsible for obtaining the bit-width of a specific operand. Moreover, syscall_entry() and syscall_exit() allow DBAs to handle system calls without much effort. The usage of the

proposed APIs is similar with the APIs provided by Pin [30], so analysts who have experiences in Pin can learn DBAF easily.

3 DBAs Based on DBAF

To demonstrate the utility of DBAF, we implement five DBAs on top of it. This section focuses on the implementation details of DBAs and shows how to use the provided APIs.

3.1 IR Counter

IR counter counts the number of executed IR, which maintains an integer representing the number of IR executed so far and increases it by one if an IR will be executed. IR counter outputs the integer when the instrumented binary finishes execution. Therefore, IR counter needs to invoke IR_InsertCall() to insert a call to a function which will be executed right before the execution of every IR. Figure 3 presents the core source code of IR counter, which uses two APIs (in bold at Line 6 and Line 12) provided by DBAF.

```
1   unsigned long long gRunInsCount = 0;
2   VOID IR_counter(ADDRINT addr) {
3     gRunInsCount += 1;
4   }
5   VOID ir_instrument_entry(IR ir, VOID* v) {
6     IR_InsertCall(ir, IPOINT_BEFORE, (AFUNPTR)IR_counter,
7       IARG_INST_PTR, IARG_END);}
8   VOID Finish(INT32 code, VOID *v) {
9     cout << "Executed IR count:" << gRunInsCount << endl;
10  }
11  int main(...){
12    IR_AddInstrumentFunction(ir_instrument_entry, 0);
13    PIN_AddFiniFunction(Finish, 0);
14    PIN_StartProgram();
15    return 0;}
```

Fig. 3. Core code of IR counter

3.2 Memory Tracer

Memory tracer records the memory address read or written by an IR and the corresponding instruction address. Figure 4 shows the core code of memory tracer. We omit the code of main() since it is the same with the main() of IR counter. Line 1 declares a file to record the trace. RecordMemRead() is responsible for recording the address read by the IR and the IR (i.e., instruction) address, which

are acquired by invoking the proposed APIs IR_ReadMemory() (Line 9) and IR_Address() (Line 7), respectively. The call to RecordMemRead() is inserted before the IR (Line 10) which reads memory (Line 8). The recording of memory write is handled in a similar way.

```
1  ofstream OutFile("trace.txt");
2  VOID RecordMemRead(ADDRINT insAddr, ADDRINT memAddr) {
3    OutFile << "0x" << hex << insAddr << ": R 0x" << hex << memAddr << endl;}
4  VOID RecordMemWrite(ADDRINT insAddr, ADDRINT memAddr) {
5    OutFile << "0x" << hex << insAddr << ": W 0x" << hex << memAddr << endl;}
6  VOID ir_instrument_entry(IR ir, VOID* v) {
7    ADDRINT insAddr = IR_Address(ir);
8    if (IR_IsMemoryRead(ir)) {
9      ADDRINT irReadMemoryAddr = IR_ReadMemory(ir);
10     IR_InsertCall(ir, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
11       IARG_UINT32, insAddr, IARG_UINT32, irReadMemoryAddr, IARG_END);
12   } else if (IR_IsMemoryWrite(ir)) {
13     ADDRINT irWriteMemoryAddr = IR_WriteMemory(ir);
14     IR_InsertCall(ir, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
15       IARG_UINT32, insAddr, IARG_UINT32, irWriteMemoryAddr, IARG_END);
   }
}
```

Fig. 4. Core code of memory tracer

3.3 Division-By-Zero Protector

Division-by-zero protector monitors the execution of a binary and halts its execution if an instruction divides zero. Figure 5 presents the core code of our division-by-zero protector (main() is omitted). The current version support unsigned integer division (UDIV_OP) and signed integer division (SDIV_OP) (Line 12). The extension for supporting floating-point division and modulo operation is straightforward. The second operand of UDIV_OP and SDIV_OP is the divisor and it can be memory (Line 16), register (Line 18) and immediate number (Line 14). The protector handles each case accordingly. The function TargetDiv() (Line 1) is responsible for checking the divisor and halting execution if the divisor is equal to zero. The call to TargetDiv() is inserted before the execution of each UDIV_OP and SDIV_OP (Line 15, 17, 19).

3.4 Taint Analyzer and Concolic Executor

Taint analysis consists of taint sources, taint propagation and taint sinks [36]. In particular, a taint analyzer marks inputs of interest (e.g., untrusted data) as taints, tracks taint propagation and takes actions (e.g., halt execution) if

```
1   void TargetDiv(int flag, void * para) {
2     bool nonzero = true;
3     if (1 == flag){//memory
4     nonzero = (*para != 0);
5     } else if (2 == flag) {//reg value or immediate number
6       nonzero = (para != 0);
      }
7     if (!nonzero) {
8       cerr << "Detected: divided by zero" << endl;
9       exit(-1);
      }
    }
10  VOID ir_instrument_entry(IR ir, VOID* v) {
11    INT opcode = IR_Opcode(ir);
12    if(opcode != UDIV_OP || opcode != SDIV_OP)
13      return;
14    if (IR_OperandIsImmediate(ir, 1)) {
15      IR_InsertCall(ir, IPOINT_BEFORE, (AFUNPTR)(TargetDiv), IARG_UINT32,
          2, IARG_PTR, (void*)IR_OperandImmediate(ir, 1), IARG_END);
16    } else if (IR_IsMemoryRead(ir)) {
17      IR_InsertCall(ir, IPOINT_BEFORE, (AFUNPTR)(TargetDiv), IARG_UINT32,
          1, IARG_UINT32, IR_ReadMemory(ir), IARG_END);
18    } else if (IR_OperandIsReg(ir, 1)) {
19      IR_InsertCall(ir, IPOINT_BEFORE, (AFUNPTR)(TargetDiv), IARG_UINT32,
          2, IARG_REG_VALUE, IR_OperandReg(ir, 1), IARG_END);
  }
}
```

Fig. 5. Core code of division-by-zero protector

taints flow into the specific place (e.g., disk files). To find taint sources and taint sinks in binaries, DBAs always instrument system calls. Our taint analyzer uses two APIs syscall_entry() and syscall_exit() to insert calls to analyst-provided functions before or after the execution of system calls. In the analyst-provided functions, taint sources are marked and actions are taken. To track taint propagation, taint analyzer conducts instrumentation in the level of IR (invoking IR_AddInstrumentationFunction()) and insert a call to a function *func* before the execution of each IR (invoking IR_InsertCall()). *func* is responsible for interpreting the semantics of IR and understand which operands should be affected by taint propagation.

Concolic execution, alias dynamic symbolic execution that runs a program concretely, tracks symbol propagation, collects constraints when encountering branches, and generates new inputs by querying a theorem prover [6]. Like taint analysis, concolic execution needs to mark symbol sources. In other words, we need to mark data of interest (e.g., test cases) as symbols. In implementation, our concolic executor instruments system calls using the provided APIs. Moreover, the concolic executor needs to track symbol propagation which is significantly

difficult than tracking taint propagation. That is because concolic execution requires elaborate interpretation of IR semantics to find how (not just which) operands are affected by symbol propagation.

Therefore, our concolic executor instruments the binary in the level of IR and interprets the semantics of each IR statement. Besides, our concolic executor leverages Z3 [26] to produce new inputs. The fact is that more complicated the instruction set, more effort should be made to implement and debug a concolic executor. The code amount of our concolic executor is 1,035 lines of C++. For comparison, Triton [34] which interprets machine instruction without IR translation has 35,120 lines of C++, including 15,698 lines of code under "Triton/src/libtriton/arch/x86" are dedicated to interpret x86 semantics [33].

4 Experiments

This section presents the results of the experiments concerning the translation overhead, followed by two practical cases about our taint analyzer.

4.1 Translation Overhead

Translation overhead is a critical factor to evaluate the efficiency of DBAF because the translation process is conducted online. We select ten benchmark programs from several well-known benchmark sets. All the benchmark programs are open source and have been used to evaluate other tools. In particular, four benchmark programs are for the purpose of I/O subsystem performance testing; two aim to evaluate the performance of memory subsystem; two evaluate CPU performance; one attempts to evaluate the performance of multi-thread and the last evaluates the performance of mutex. The purpose and code amount of those benchmark programs are presented in Table 2. Please note that SysBench is an integrated benchmark, and SysBench1, SysBench2, SysBench3, SysBench4, SysBench5 indicate its different functionalities. For the same reason, we do not count the code amount of those five benchmark programs separately.

To accurately measure translation overhead, we implement two versions of NullTool (termed by NullPin and NullDBAF) which are directly built on top of Pin and DBAF, respectively. NullPin just loads the benchmark programs into memory and runs them without instrumentation. NullDBAF loads the benchmark programs, translates machine instructions into IR, then converts back to instructions and runs the programs. We measure the execution time of each benchmark program loaded by NullPin and NullDBAF, respectively and then we compute the overhead as shown in Fig. 6. The overhead averaged from the ten benchmark programs is about 4x. We need to remind that the results are conservative because NullTool does not instrument the benchmark programs. Imaging the DBAs with practical functionalities, the instrumentation overhead should be much higher than translation overhead. For example, a concolic executor often slows down the execution of analyzed programs hundreds of times. Therefore, the translation overhead incurred by DBAF is reasonable. We plan to find methods to further reduce translation overhead in our future work.

Table 2. Benchmark programs to evaluate translation overhead

Benchmark	Purpose	Code amount
Bonnie++[a]	I/O	2,919
fs_mark[b]	I/O	1,067
IOzone[c]	I/O	26,681
mbw[d]	memory	207
stress[e]	CPU	628
SysBench1[f]	CPU	7,452
SysBench2	Memory	7,452
SysBench3	Multi-thread	7,452
SysBench4	Mutex	7,452
SysBench5	I/O	7,452

https://sourceforge.net/projects/bonnie/.
https://github.com/josefbacik/fs_mark.
http://www.iozone.org/.
https://github.com/raas/mbw.
http://people.seas.harvard.edu/~apw/stress/.
https://github.com/nuodb/sysbench.

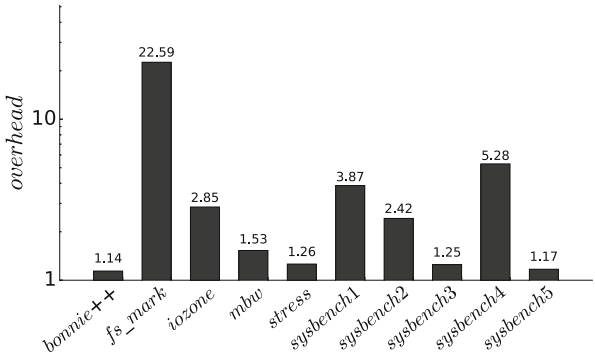

Fig. 6. Translation overhead of `DBAF`

4.2 Practical Cases

We evaluate the effectiveness of our taint analyzer through validating two practical vulnerabilities. To speedup the validation process, we record the memory range of the analyzed binary and restrict instrumentation in this range. In other words, we do not process the code of libraries.

CVE-2010-4051. This vulnerability exists in the function regcomp() of the GNU C library that processes untrusted inputs without preliminary checking the

input for the sanity [9]. Consequently, attackers can craft an exploit containing adjacent bounded repetitions (e.g., {10,}{10,}{10,}{10,}{10,}) to trigger a stack overflow in regcomp(), resulting in a crash. To detect various kinds of control-flow hijacking attacks (including stack overflow), we enrich the taint sinks of our taint analyzer. In particular, we consider all indirect rets/jumps/calls as taint sinks, and therefore our taint analyzer detects a control-flow hijacking if the target of an indirect ret/jump/call is tainted. Please note that direct jumps/calls are not included in the taint sinks because attackers cannot subvert the jump/call targets. Our taint analyzer instruments 135,082 IR (corresponding to 21,652 instructions) and detects the vulnerability in 68 s. Before triggering the bug, 7,154,041,604 IR (corresponding to 713,671,310 instructions) are executed.

CVE-2010-0001. This vulnerability results from an integer overflow in function unlzw() of gzip before 1.4 on 64-bit platforms, allowing remote attackers to launch a DoS attack or possibly execute arbitrary code [8]. The outcome of the overflowed integer computation is used as an array index, and hence attackers control the array index and then possibly get access to arbitrary memory address. To detect memory corruption, we enrich the taint sinks of our taint analyzer. In particular, we consider all memory operations (i.e., load, store) as taint sinks, and hence our taint analyzer detects an attack for memory corruption if the binary gets access to a tainted address. Our taint analyzer instruments 109,913 IR (corresponding to 17,298 instructions) and detects the vulnerability in 170 s. Before triggering the flaw, 21,875,318,127 IR (corresponding to 2,956,988,800 instructions) are executed.

5 Related Work

This section reviews studies about binary instrumentation tools, rather than the applications based on binary instrumentation tools. Pin [24], Dyninst [1] and DynamoRIO [2] are three well-known dynamic instrumentation tools (DITs) that have been widely used in both academia and industry. Besides, new DITs usually either build on top of them or compare with them. Pin [24] is developed by Intel Corp. that is efficient and provides rich APIs and well-written documents. Dyninst [1] supports both dynamic instrumentational static instrumentation (i.e., binary rewrite) and provides unified APIs for both. DynamoRIO attempts to construct a transparent instrumentation environment because the behaviors of the instrumented binary may be changed if it is aware of the fact that it is running in an instrumentation environment [3].

To reduce the runtime overhead of the binary after instrumentating by a static instrumentation tool (SIT), PEBIL uses function level code relocation in order to insert large but fast control structures and then allows analysts to insert assembly code directly [21]. Compared with a DIT, a binary after processing by a SIT has lower runtime overhead, however, the instrumentation of SITs is easy to be bypassed. Consequently, SITs are less commonly-used to analyze malware. PSI enhances SITs by ensuring a non-bypassable instrumentation [45].

In particular, PSI enforces three properties to achieve the non-bypassability, e.g., all direct and indirect control-ow transfers made from the original code must target instructions in the original code that were validly disassembled by the disassembler [45].

EEL proposes a RISC-like IR, allowing analysts to write machine- and OS-independent applications [18]. Strata is a dynamic instrumentation framework supporting SPARK and MIPS instruction sets [37]. Unlike EEL, Strata does not translate machine instructions into IR possibly because SPARK and MIPS are RISC instructions sets. Vulcan supports both dynamic and static instrumentation, which translates instructions into MSIL (an IR designed by Microsoft Corp.) and provides APIs [11]. Hazelwood and Klauser extends Pin to support ARM instruction set [16]. Dimension is a DIT for virtual execution environments (VEEs) that has two advantages in design [44]. First, Dimension is not tightly coupled with VEE, so it can be reused easily be different VEEs. Besides, it is able to instrument both source and target binaries. HDTrans is a light-weight DIT designed for those binaries with a small and hot working set [39]. DSPInst is a SIT for Blackfin DSP processor [40], DPCL is the extension of Dyninst for supporting parallel MPI applications [22,35] and PMaCinst is a SIT supporting PowerPC instruction set [41].

Guillon proposes to instrument binary via QEMU, a cross-platform emulation tool, in order to instrument the entire software stack, including kernel modules [14]. For the similar purpose, PinOS leverages XEN, a virtual machine hypervisor to extends Pin with the ability to instrument the whole operating system [4]. Technically, PinOS runs under the guest OS to manipulate the guest OS. Feiner et al. propose a different design which implements a Linux kernel module to conduct a whole-system instrumentation [12].

Mobile devices are weaker than desktop computers in terms of processing/memory/storage capability. SIF is a selective instrumentation framework for mobile applications, enabling analysts to specify a small amount of code in applications to be instrumented, thus overhead on mobile devices can be reduced [15]. DIOTA circumvents the challenges of constructing a control flow graph and enables to instrument self-modifying code [25]. VMAD first instruments the source code of the analyzed software by LLVM and then monitors its execution in a virtual machine [17]. SecondWrite is a SIT that is able to instrument stripped binaries (i.e., without relocation information) [38]. It is a technical challenge for DITs to instrument multi-thread programs. Chung et al. apply transactional memory to enclose the data and metadata accesses within an atomic transaction, thus thread safe is maintained [7]. DIABLO is a static instrumentation framework, which translates various instruction sets into IR and provides APIs [32].

To overcome the limitation of static disassembling, BIRD combines static disassembly with an on-demand dynamic disassembly approach to guarantee that each instruction in a binary file is analyzed or transformed before it is executed [27]. SuperPin proposes to speedup instrumentation by dividing the analyzed binary into non-overlapped instruction sequences, and then starts multiple instrumentation threads to process each sequence in parallel [43]. Upton and

Cohn observe that both data collection and data analysis of binary instrumentation are time-consumption. They propose to decouple data collection from analysis and buffer the data for analysis [42]. To ease its usage for analysts, Hijacker proposes rule-based instrumentation that allows analysts to write instrumentation requirements in an xml file [29]. Our previous work designs a middleware to take care of the differences of various instrumentation tools and expose easy-to-use APIs to analysts [5]. However, the middleware does not translate instructions into IR, so analysts have to interpret instruction semantics by themselves.

6 Conclusion

Dynamic binary instrumentation is a fundamental technique for various applications. Existing DITs have their shortcomings. This study design DBAF, a dynamic binary analysis framework that translates machine instructions into LLVM IR and provides tens of Pin-like APIs enabling analysts to instrument the binary in the level of IR easily. Moreover, we present five applications based on DBAF. Experiments show that the translation overhead is reasonable. We will try to further reduce the translation overhead in our future work.

Acknowledgment. This work is supported in part by National Key R&D Program of China (2017YF-B0802903), Project 2117H14243A and Sichuan Province Research and Technology Supporting Plan, China.

References

1. Bernat, A., Miller, B.: Anywhere, any-time binary instrumentation. In: PASTE (2011)
2. Bruening, D., Duesterwald, E., Amarasinghe, S.: Design and implementation of a dynamic optimization framework for windows. In: FDDO (2001)
3. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: VEE (2012)
4. Bungale, P.P., Luk, C.K.: Pinos: a programmable framework for whole-system dynamic instrumentation. In: VEE (2007)
5. Chen, T., Xu, Y., Zhang, X.: A program manipulation middleware and its applications on system security. In: Lin, X., Ghorbani, A., Ren, K., Zhu, S., Zhang, A. (eds.) SecureComm 2017. LNICST, vol. 238, pp. 606–626. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78813-5_31
6. Chen, T., Zhang, X., Guo, S., Li, H., Wu, Y.: State of the art: dynamic symbolic execution for automated test generation. Future Gener. Comput. Syst. **29**(7), 1758–1773 (2013)
7. Chung, J., Dalton, M., Kannan, H., Kozyrakis, C.: Thread-safe dynamic binary translation using transactional memory. In: HPCA (2008)
8. CVE: Cve-2010-0001 (2011). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0001
9. CVE: Cve-2010-4051 (2011). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4051

10. Dinaburg, A., Adve, V.: McSema: static translation of x86 instructions to LLVM. In: ReCon (2014)
11. Edwards, A., Vo, H., Srivastava, A.: Vulcan binary transformation in a distributed environment (2001). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2001-50.pdf
12. Feiner, P., Brown, A.D., Goel, A.: Comprehensive kernel instrumentation via dynamic binary translation. In: ASPLOS (2012)
13. Feng, Y.: Fixed potential LLVM value type dismatch in llvm::constantint::get. (#241) #242 (2017). https://github.com/trailofbits/mcsema/pull/2421
14. Guillon, C.: Program instrumentation with QEMU. In: International QEMU Users' Forum (2011)
15. Hao, S., Li, D., Halfond, W.G., Govindan, R.: SIF: a selective instrumentation framework for mobile applications. In: Mobisys (2013)
16. Hazelwood, K., Klauser, A.: A dynamic binary instrumentation engine for the arm architecture. In: CASES (2006)
17. Jimborean, A., Mastrangelo, L., Loechner, V., Clauss, P.: VMAD: an advanced dynamic program analysis and instrumentation framework. In: O'Boyle, M. (ed.) CC 2012. LNCS, vol. 7210, pp. 220–239. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28652-0_12
18. Larus, J.R., Schnarr, E.: EEL: machine-independent executable editing. In: PLDI (1995)
19. Lattner, C.: The design of LLVMS (2012). http://www.drdobbs.com/architecture-and-design/the-design-of-llvm/240001128?pgno=1
20. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO (2004)
21. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snavely, A.: PEBIL: efficient static binary instrumentation for Linux. In: ISPASS (2010)
22. Lee, G.L., et al.: Dynamic binary instrumentation and data aggregation on large scale systems. Int. J. Parallel Program. **35**(3), 207–232 (2007)
23. LLVM: LLVM language reference manual (2018). https://llvm.org/docs/LangRef.html
24. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI (2005)
25. Maebe, J., Ronsse, M., Bosschere, K.D.: Diota: dynamic instrumentation, optimization and transformation of applications. In: WBT (2002)
26. Moura, L.D., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS (2008)
27. Nanda, S., Li, W., Lam, L.C., Chiueh, T.C.: Bird: binary interpretation using runtime disassembly. In: CGO (2006)
28. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI (2007)
29. Pellegrini, A.: Hijacker: efficient static software instrumentation with applications in high performance computing: poster paper. In: HPCS (2013)
30. Pin: API reference (2017). https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/group__API__REF.html
31. Prasad, M.: Disassembly challenges (2003). http://static.usenix.org/event/usenix03/tech/full_papers/prasad/prasad_html/node5.html
32. Put, L.V., Chanet, D., Bus, B.D., Sutter, B.D., Bosschere, K.D.: DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In: ISSPIT (2005)
33. Salwan, J.: Triton source code (2018). https://github.com/JonathanSalwan/Triton/tree/master/src/libtriton/arch/x86

34. Saudel, F., Salwan, J.: Triton: concolic execution framework (2015). http://shell-storm.org/talks/SSTIC2015_English_slide_detailed_version_Triton_Concolic_Execution_FrameWork_FSaudel_JSalwan.pdf
35. Schulz, M., et al.: Scalable dynamic binary instrumentation for blue gene/l. In: WBIA (2005)
36. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: S&P (2010)
37. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: CGO (2003)
38. Smithson, M., Anand, K., Kotha, A., Elwazeer, K., Giles, N., Barua, R.: Binary rewriting without relocation information (2010). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.463.3748&rep=rep1&type=pdf
39. Sridhar, S., Shapiro, J.S., Northup, E., Bungale, P.P.: HDTrans: an open source, low-level dynamic instrumentation system. In: VEE (2006)
40. Sun, E., Kaeli, D.: A binary instrumentation tool for the blackfin processor. In: WBIA (2009)
41. Tikir, M.M., Laurenzano, M., Carrington, L., Snavely, A.: PMAC binary instrumentation library for powerpc/aix. In: WBIA (2006)
42. Upton, D., Hazelwood, K., Cohn, R., Lueck, G.: Improving instrumentation speed via buffering. In: WBIA (2009)
43. Wallace, S., Hazelwood, K.: Superpin: parallelizing dynamic instrumentation for real-time performance. In: CGO (2007)
44. Yang, J., Zhou, S., Soffa, M.L.: Dimension: an instrumentation tool for virtual execution environments. In: VEE (2006)
45. Zhang, M., Qiao, R., Hasabnis, N., Sekar, R.: A platform for secure static binary instrumentation. In: VEE (2014)